**DATA STRUCTURES AND ALGORITHMS**

**2. E-commerce Platform Search Function**

Big O Notation is a mathematical representation used to describe the efficiency or complexity of an algorithm in terms of:

* Time complexity: how the running time increases as input size grows.
* Space complexity: how much memory the algorithm uses as input size increases.

->Purpose of Big O:

* It helps in comparing algorithms independently of hardware or programming language.
* Focuses on the worst-case scenario, which is important for ensuring performance under pressure.
* Ignores constants and lower-order terms to focus on growth trends.

**CODE:**

import java.util.\*;

public class Product

{

public static void main(String args[])

{

int i, c = 0, lb, ub, mid;

int pid[] = new int[100];

String cat[] = new String[100];

String pname[] = new String[100];

Scanner sc = new Scanner(System.in);

System.out.print("Enter the number of products = ");

int n = sc.nextInt();

sc.nextLine();

System.out.println("Enter the product ID's of the products:");

for (i = 0; i < n; i++)

{

pid[i] = sc.nextInt();

sc.nextLine();

}

System.out.println("Enter the categories of the products:");

for (i = 0; i < n; i++)

{

cat[i] = sc.nextLine();

}

System.out.println("Enter the product names:");

for (i = 0; i < n; i++)

{

pname[i] = sc.nextLine();

}

int choice;

do {

System.out.println("\n1. Linear Search by Product ID");

System.out.println("2. Linear Search by Product Name");

System.out.println("3. Linear Search by Product Category");

System.out.println("4. Binary Search by Product ID");

System.out.println("5. Binary Search by Product Name");

System.out.println("6. Binary Search by Product Category");

System.out.println("7. Exit");

System.out.print("Enter your choice: ");

choice = sc.nextInt();

sc.nextLine();

switch (choice) {

case 1:

c = 0;

System.out.print("Enter the product ID to be searched: ");

int s = sc.nextInt();

sc.nextLine();

for (i = 0; i < n; i++) {

if (pid[i] == s) {

c = 1;

break;

}

}

if (c == 1)

System.out.println("Product is found");

else

System.out.println("Product is not found");

break;

case 2:

c = 0;

System.out.print("Enter the product name to be searched: ");

String s1 = sc.nextLine();

for (i = 0; i < n; i++) {

if (pname[i].equals(s1)) {

c = 1;

break;

}

}

if (c == 1)

System.out.println("Product is found");

else

System.out.println("Product is not found");

break;

case 3:

c = 0;

System.out.print("Enter the product category to be searched: ");

String s2 = sc.nextLine();

for (i = 0; i < n; i++) {

if (cat[i].equals(s2)) {

c = 1;

break;

}

}

if (c == 1)

System.out.println("Product is found");

else

System.out.println("Product is not found");

break;

case 4:

c = 0;

Arrays.sort(pid, 0, n);

System.out.print("Enter the product ID to be searched: ");

s = sc.nextInt();

sc.nextLine();

lb = 0;

ub = n - 1;

while (lb <= ub) {

mid = (lb + ub) / 2;

if (pid[mid] == s) {

c = 1;

break;

} else if (pid[mid] < s)

lb = mid + 1;

else

ub = mid - 1;

}

if (c == 1)

System.out.println("Element found");

else

System.out.println("Element not found");

break;

case 5:

c = 0;

Arrays.sort(pname, 0, n);

System.out.print("Enter the product name to be searched: ");

s1 = sc.nextLine();

lb = 0;

ub = n - 1;

while (lb <= ub) {

mid = (lb + ub) / 2;

if (pname[mid].equals(s1)) {

c = 1;

break;

} else if (pname[mid].compareToIgnoreCase(s1) < 0)

lb = mid + 1;

else

ub = mid - 1;

}

if (c == 1)

System.out.println("Element found");

else

System.out.println("Element not found");

break;

case 6:

c = 0;

Arrays.sort(cat, 0, n);

System.out.print("Enter the product category to be searched: ");

s2 = sc.nextLine();

lb = 0;

ub = n - 1;

while (lb <= ub) {

mid = (lb + ub) / 2;

if (cat[mid].equals(s2)) {

c = 1;

break;

} else if (cat[mid].compareToIgnoreCase(s2) < 0)

lb = mid + 1;

else

ub = mid - 1;

}

if (c == 1)

System.out.println("Element found");

else

System.out.println("Element not found");

break;

case 7:

System.out.println("Exiting...");

break;

default:

System.out.println("Invalid choice!");

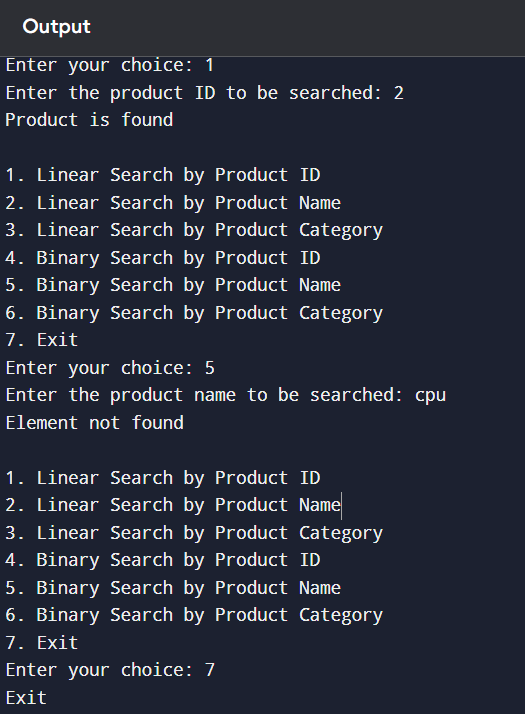
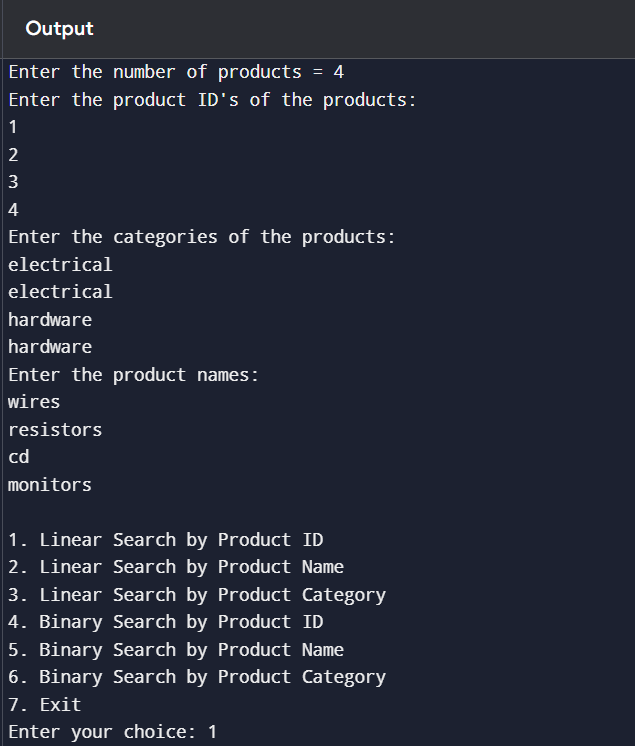
}

} while (choice != 7);

}

}

**OUTPUT:**

****

Linear search checks each element in the list one by one.  
If the element is at the start, it finds it quickly (best case).  
If it’s at the end or not present, it has to check everything (worst case).  
So its worst-case time complexity is O(n) where n is the number of elements.

Binary search works differently. It only works on sorted data.  
It repeatedly divides the list in half and checks the middle value.  
This cuts the problem size down quickly, making it much faster.  
Its worst-case time complexity is O(log n).

So, binary search is much more efficient than linear search for large lists, but only if the list is sorted.

For our case as we have been told to design on the basis of both binary search and linear search both can be Binary search for sorted data and linear search for unsorted data . However Binary search can be more suitable for building large scale applications where the main goal is to reduce the time for searching.

* 1. **FINANCIAL FORECASTING**

RECURSION: Recursion is when a function calls itself to solve a smaller version of a problem.

Instead of solving the entire problem at once, recursion breaks it down into smaller and smaller subproblems until it reaches something simple — called the base case — and then it works its way back up.

Some problems are naturally recursive. That means they’re made up of repeating smaller versions of themselves. Recursion helps solve them with cleaner and shorter code, especially when the structure of the problem is repetitive .

**CODE:**

import java.util.\*;

public class finance

{

static double financecalc(int years,double previousvalue,double rate)

{

if(years==1||years==0)

return previousvalue;

else

return financecalc(years-1,previousvalue\*(1+rate/100),rate);

}

public static void main(String args[])

{

Scanner sc = new Scanner(System.in);

double result;

System.out.println("Enter the number of years upto which you want to calculate");

int years=sc.nextInt();

System.out.println("Enter the previous value");

double previousvalue=sc.nextDouble();

System.out.println("Enter the rate ");

double rate=sc.nextDouble();

result=financecalc(years,previousvalue,rate);

if(years<0)

System.out.println("Calculation not possible");

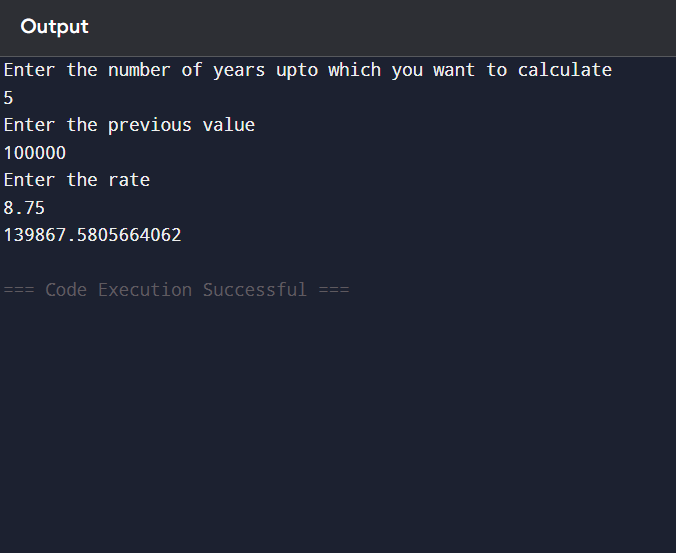
else

System.out.println(result);

}

}

**OUTPUT:**



Time complexity of my algorithm: O(n)

How to optimize: Since this is a compound Interest formula we can use the math library for direct power and return the value in O(1) time.